AFIT/GCS/ENG/99M-22

TRANSFORMING COBOL LEGACY SOFTWARE TO A GENERIC IMPERATIVE MODEL

THESIS

Diná Leite Moraes, Captain, Brazilian Air Force

AFIT/GCS/ENG/99M-22

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
TRANSFORMING COBOL LEGACY CODE SOFTWARE TO A GENERIC IMPERATIVE MODEL

5. FUNDING NUMBERS

6. AUTHOR(S)
Dina Leite Moraes, Captain, Brazilian Air Force

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/99M-22

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Roy F. Stratton
AFRL/IFTD
525 Brooks Rd.
Rome, NY 13441-4505
(303) 315-3004 (DSN 587-3004)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Robert P. Graham, Jr.
Robert.graham@afit.af.mil
(513) 255-9918

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Legacy COBOL code presents challenges for organizations wishing to increase their effectiveness and improve their efficiency. The systems that utilize this code are vital to enterprise operations, yet maintenance of code is expensive. This research presents an alternative that overcomes some of the problems of legacy COBOL code through the use of software reengineering techniques.

This research develops a transformation system to convert COBOL code into a generic imperative model, recapturing the initial design and deciphering the requirements implemented by the legacy code, thereby making possible the documentation of legacy systems, design restructuring, and the re-design of a new, enhanced system.

14. SUBJECT TERMS
COBOL, Generic Language, Legacy Software, Reengineering

15. NUMBER OF PAGES
89

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

The views expressed in this thesis are those of the author and do not reflect the official policy or position of Ministerio da Aeronautica do Brasil.

AFIT/GCS/ENG/99M-22

TRANSFORMING COBOL LEGACY SOFTWARE
TO A GENERIC IMPERATIVE MODEL

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Diná Leite Moraes, B.S.

Captain, Brazilian Air Force

March 1999

AFIT/GCS/ENG/99M-22

TRANSFORMING COBOL LEGACY SOFTWARE TO A GENERIC IMPERATIVE MODEL

Diná Leite Moraes, B.S.

Captain, Brazilian Air Force

Approved:

Robert P. Graham Jr — 4 MAR 99

Dr. Robert P. Graham, Jr., Chairman — date

Thomas C. Hartrum — 4 Mar 1999

Dr. Thomas C. Hartrum — date

Scott — 4 Mar 99

Dr. Scott A. DeLoach — date

Acknowledgements

I am extremely grateful to the Brazilian Air Force for giving me the opportunity to complete an AFIT degree. There is no doubt I have developed additional skills that will serve me throughout my life. I extend my deep appreciation and admiration to the faculty and staff of the Air Force Institute of Technology, School of Engineering for helping me to attain my goals and for their dedication and enthusiasm in teaching and research. In particular, I'd like to thank my advisor, Maj. Robert P. Graham, Jr. for his guidance and support throughout the course of this thesis effort.

I would like also to express my appreciation to my Brazilians fellows, Capt. Sonia Rodrigues and Maj. Ferreira Gomes and his family. They made me feel as comfortable here as I would have been in my home country. I would also like to thank Mrs. Annette Robb for the consistent and heartfelt support she gave not only to me but also to all international students.

Finally, I'd like to express my deep appreciation to my fiancee Capt Joseph Henry Scherrer. His love and faith in my abilities gave me the confidence I needed to overcome all obstacles in this difficult process. The relationship we built during our time here at AFIT provided us with the foundation for continual happiness, joy, commitment, and strength as we embark on our life together.

Diná Leite Moraes

Table of Contents

Acknowledgements iii

List of Figures vii

List of Tables viii

Abstract ix

I. Introduction 1

Chapter Overview 1

Background 1

Research Context 3

Problem Statement 5

Thesis Outline 6

II. Literature Review 7

Chapter Overview 7

Generic Programming Languages and the Generic Imperative Model 7

Transformation Systems 11

TAMPR transformation system. 12

Inverse Transformation. 13

COBOL Program Reuse. 16

Draco-PUC Transformational Engine. 18

Other Techniques for Extracting Specifications from COBOL Code 19

Creating Specifications from Code. 19

Reverse Engineering from COBOL to Narrative Specification. 19

Summary. 21

III. Methodology 22

Chapter Overview 22

Methodology 22

GIM Restrictions 27

Restriction 1: 27

Restriction 2: 27

Restriction 3: 28

Restriction 4: 28
Restriction 5: 28
Restriction 6: 28
Restriction 7: 29
Restriction 8: 29
Restriction 9: 29

COBOL Constructs Not Implemented 29
Group 1.......... 30
Group 2.......... 30
Group 3.......... 31
Group 4.......... 32
Group 5.......... 32

Summary 33

IV. Transformation 34

Chapter Overview 34

Introduction 34

COBOL Add Transform.......... 36

COBOL Subtract Transform. 38

COBOL Move Transform. 40

COBOL Redefines Transform. 41

Record Elimination. 41

Summary 44

V. Translation and Analysis of the GIM 45

Chapter Overview 45

Translation of COBOL constructs.......... 45
Add Statement. 45
Subtract Statement.......... 47
Multiply Statement. 47
Divide Statement. 47
Compute Statement. 48
Move Statement.......... 48
Accept Statement.......... 49
Read Statement.......... 49
Display Statement. 50
Write Statement.......... 51
Open Statement. 52
If Statement. 52
Call Statement. 53
Perform Statement.......... 54

The Program. 57

Building the new GIM AST 58

New COBOL Attributes. 59

Analysis of the GIM 60

Summary 65

VI. Conclusion 66

Analysis of the Results 66

Recommendations 69

Future Research 70

Summary 70

Bibliography 71

Appendix A: Example of COBOL Code Transformation 73

Vita 79

List of Tables

Table Page

1. COBOL Constructs Translated into the GIM 46
2. COBOL Constructs Translated 67
3. COBOL Constructs Not Implemented 68

List of Figures

Figure Page

1. Reengineering Process 2
2. Reverse Engineering Process 4
3. GIM Domain Model Inheritance Hierarchy 11
4. Reverse Engineering and Restructuring 14
5. COBOL Code Transformation and its Translation 26
6. COBOL Add Statement AST 35
7. Imperative-Input 50
8. Imperative-Output 51
9. Imperative-File 52
10. Imperative-Selection 53
11. Imp-Subprogram-Call 56
12. Imperative-AST 58
13. Imperative-Record 61
14. Revised Imperative-Name 62
15. Imperative-Data-Type 63
16. Imperative-File Representing Ordered Files 65
17. COBOL Code Before Transformation 74
18. COBOL code After Transformation 76
19. COBOL Code After Translation 78

AFIT/GCS/ENG/99M-22

Abstract

Legacy COBOL code presents challenges for organizations wishing to increase their effectiveness and improve their efficiency. The systems that utilize this code are vital to enterprise operations, yet maintenance of code is expensive. This research presents an alternative that overcomes some of the problems of legacy COBOL code through the use of software reengineering techniques.

This research develops a transformation system to convert COBOL code into a generic imperative model, recapturing the initial design and deciphering the requirements implemented by the legacy code, thereby making possible the documentation of legacy systems, design restructuring, and the re-design of a new, enhanced system.

TRANSFORMING COBOL LEGACY SOFTWARE TO A GENERIC IMPERATIVE MODEL

I. Introduction

Chapter Overview

This chapter discusses the status of legacy systems in the Brazilian Air Force and efforts to keep them operational and updated. Next, reverse engineering is explained as a possible method to aid in the renovation of these legacy systems. Finally, a reverse engineering method for the purpose of this research is described.

Background

Despite new information technologies that exist in the Brazilian Air Force for developing computer systems, many systems still use early versions of third generation languages, in particular COBOL. These so-called "legacy systems" are large systems that were developed years ago using now-obsolete technology but that are still reliable for the organizations. They represent years of accumulated experience and knowledge, and often are the only place where the organization's business rules can be found. These systems are very hard to maintain because they are poorly structured, poorly documented, or both. Despite these problems, legacy systems are vital for administration and operation of Brazilian Air Force. Based mainly on the analysis of these legacy systems, alternative software has been developed to replace them. This analysis is done by analyzing COBOL program source code, because documentation of the original program code is outdated and unreliable. In order to replace an old system with a new one, the new system must perform at least as well as the legacy system. Evaluating these results without formal documentation of the user requirements is a very hard task. Some activities modeled and implemented in the legacy system are not frequently executed, and for this reason it is difficult to detect that they are missing in the new system. To avoid the lack of some important activity in the new system, the transition between systems is made during a long period of time, demanding the users do the same activities in both systems. Even with all these precautions there is still uncertainty that the new system is achieving at least all the old system results. To avoid chaos in the organization, in most cases, these legacy systems are never discontinued.

Reengineering of legacy code may provide a way to avoid this chaos. Reengineering, also known as both renovation and reclamation of legacy code, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [4]. Figure 1 shows a generalized view of the process of reengineering legacy code as developed by Byrne [3].

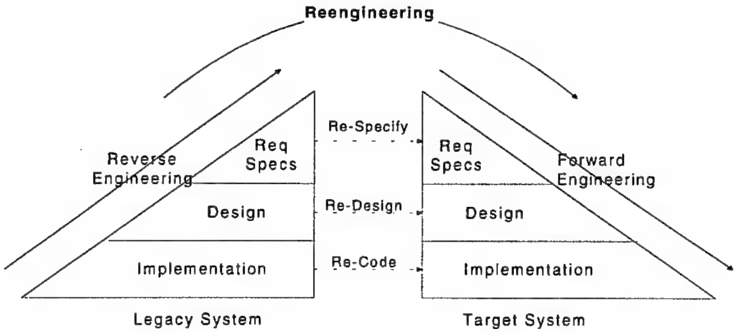


Figure 1. Reengineering Process

In order to effectively reengineer the legacy code, it must be expressed at a higher level of abstraction than the programming language in which it was written [15]. This process of expressing the legacy code in a higher level of abstraction is ***reverse engineering***. The different levels of abstraction in reengineering include implementation, design, and requirements specification [13]. Reverse engineering does not involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of examination, not a process of change or replication [4].

Legacy code can be reengineered at each of these levels of abstraction. At the implementation level, it is possible to re-code a program from one programming language to another. At the design level, it is possible to re-design a program changing the design of the legacy code into a design for the target system. At the requirement specification level, it is possible to re-specify the requirements for a program [13].

Research Context

Legacy systems with missing or inaccurate documentation are very difficult to operate and maintain because the primary source of information for these systems is the source code itself. This old code cannot be easily understood because of the following reasons:

- With a third generation language such as COBOL it is very difficult to use structured programming techniques. This makes the program logic unapparent and consequently hard to understand.
- Over the years software modifications make the software even more complex and less maintainable and its documentation completely outdated. This is

called "program erosion" and is the result of software modifications implemented unsystematically and without concern for the initial design.

- Data structures used in these programs are affected by the small main memory.
- These programs have typically many lines of code that make their understanding more difficult.
- They have no formal documentation.

Given these issues, reverse engineering could be the part of the maintenance process that would help understand the legacy systems to make the appropriate changes starting from the existing implemented code and recapturing or recreating the design and deciphering the requirements actually implemented by the subject system [4]. Figure 2 shows the process of reverse engineering in more detail:

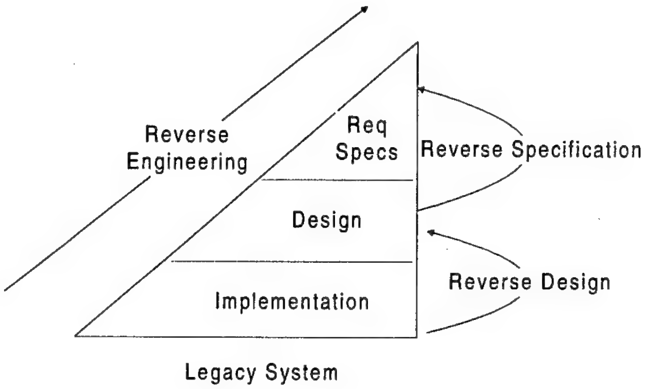


Figure 2. Reverse Engineering Process

The process of reverse design is the focus of this research. Reverse design abstracts the implementation information up to the design level. This process extracts information such as a structure chart showing the calling hierarchy of the legacy code, data flow diagrams showing the flow of data in and between legacy code routines, or control flow diagrams showing the flow of control for the legacy code.

One method of reverse design developed by Sward includes as its first step transforming the legacy code into a canonical form [13]. A canonical form allows code that performs the same to look the same. Sward's reengineering methodology uses the Generic Imperative Model (GIM) as the canonical form. The GIM includes fundamental aspects of imperative programming languages. The imperative programming language constructs are modeled in the GIM by building abstract syntax trees that store knowledge about the constructs.

This thesis describes the transformations necessary to translate a representative sample of the Brazilian Air Force Maintenance and Supply System COBOL source code to the GIM and evaluate the ability of Sward's Generic Imperative Model to handle the COBOL language in the reverse engineering process at the design level. This transformation system was developed in partnership with Capt Sonia Rodrigues (Brazilian Air Force).

Problem Statement

Legacy COBOL code presents challenges for organizations wishing to increase their effectiveness and improve their efficiency. The systems that use this code are vital to enterprise operations, yet maintenance of code is expensive. This research presents an alternative that overcomes some of the problems of legacy COBOL code through the use of software reengineering techniques. Using a commercial off-the-shelf software package called Refine/COBOL^{TM}, part of Software Refinery ^{TM} , a representative sample of Brazilian Air Force legacy COBOL code is reverse engineered. The reverse engineering process uses Refine/COBOL^{TM} to transform legacy COBOL code into the Generic Imperative Model (GIM). This process uses the GIM to express the COBOL legacy code in a higher level of abstraction. After this transformation, the COBOL legacy code is modularized and represented through a collection of imperative subprograms where the constructs that perform the same function have the same syntax.

This research aids in the understanding of legacy code, thereby making possible the documentation of legacy systems, the extension of vital business functions, the implementation of new functions, and the generation of new enhanced source code that captures the key aspects of the original code.

Thesis Outline

Chapter II of this paper will present a review of applicable literature from technical sources. Chapter III will provide the methodology to develop the transformation system outlined in Chapter I. Chapter IV describes the implementation of this system. Finally, a discussion of the results and subsequent conclusions will be included in Chapter V.

II. Literature Review

Chapter Overview

It is well accepted by the software engineering community that one of the more important aspects of reengineering is to move COBOL code to a higher abstraction level, i.e., to reverse engineer COBOL, because the presentation at a higher abstraction level is more easily understood [7]. Due to the rising popularity of such reengineering efforts, there are quite a number of papers available that discuss methods and techniques that focus on COBOL. This chapter examines some of the papers that focus on getting a description of COBOL code at a higher level of abstraction as the first step of a reverse engineering effort.

First, an approach advocating generic language technology as a prerequisite to building analysis and reengineering tools, such as Sward's Generic Imperative Model, is discussed. Next, methodologies to reverse engineer COBOL programs to their design using transformation systems are described. Finally, techniques to create specifications from COBOL code that involve human expertise are outlined.

Generic Programming Languages and the Generic Imperative Model

According to van den Brand, Klint, and Verhoef, generic language technology is a prerequisite to building analysis and conversion tools that are needed for the reengineering of large software systems [9]. They argue that since many legacy systems are polylingual it is important that reengineering systems are based on generic language technology.

Legacy code or even complete systems are written in numerous dialects of "old-fashioned" programming languages that have to be understood and analyzed. Developing new tools for all dialects is far too expensive and can be done more effectively using generic techniques. Van den Brand, Klint, and Verhoef claim that a system is language-generic if the language can be a parameter of the system and upon instantiation with a language definition a language-specific system is obtained.

By this definition, Sward's Generic Imperative Model (GIM) can be viewed as a generic model that upon instantiation can generate a specific imperative language system. Sward developed the Parameter-Based Object Identification (PBOI) methodology for automatically extracting functionally equivalent object-oriented designs from legacy imperative programs [13]. The PBOI methodology is based on fundamental ideas that relate programs written in imperative languages to objects and classes written in object-oriented languages. To focus this task, Sward developed the Generic Imperative Model (GIM) as a generic model of imperative programming languages, and the Generic Object-Oriented Design Model (GOM) as a generic model of object-oriented programming languages. Using formal transformations, imperative subprograms represented in the GIM are converted into classes and objects represented in the GOM.

The GIM is a generic language that provides a canonical form for representing imperative programs. According to Sward, a canonical form must be:

– *language independent* - the representation is not tied to any one specific programming language;

- *programming construct independent* - constructs that provide the same control flow, even having a different syntax, can be recognized as the same entity;
- *control flow construct independent* - equivalent control structures can be represented as the same control flow entity.

The GIM includes fundamental aspects of imperative programming languages. An imperative programming language paradigm is a style of programming based on the following concepts:

- *Variables* - Variables hold state information during execution of the program;
- *Data Types* - Data types define the acceptable values for a variable and the operations that can be done on the variable
- *Expressions* - Expressions are combinations of variables and operations used to express temporary intermediate values;
- *Assignment Statements* - Assignment statements change state by assigning new values to variables via expression evaluation;
- *Input/Output* - Input and output statements read and write to the standard input/output devices and to files;
- *Sequential Control* - In sequential control flow, a sequence of statements executes one after another;
- *Selective Control* - In selective control flow, a choice is made, based on the result of a boolean expression, between executing one sequence of statements versus another;
- *Iterative Control* - In iterative control flow, a sequence of statements is executed repeatedly while a boolean expression is true;

- *Procedural Abstraction* - A procedural abstraction collects a sequence of statements that are executed when the abstraction is referenced by name. A procedural abstraction can be passed parameters and may return values.
- *Main Program* - In systems of imperative subprograms, there is always one subprogram that is given the flow of control as the system begins execution. This special subprogram is termed the main program.

Imperative programming languages include FORTRAN, C, Pascal, Ada, COBOL, and any language that implements the concepts presented above. These imperative programming language constructs are modeled in the GIM by building abstract syntax trees that store knowledge about the constructs. For each programming language construct modeled in the GIM, formal semantics are provided using the state model of programs. Pre conditions and post conditions are used to define the semantics for each GIM representation of an imperative construct. Specifically, given a post condition R that is guaranteed to be true after a statement S is executed, the weakest precondition, wp(S,R), defines the weakest set of preconditions that must hold in order for the execution of S to establish R.

Figure 3 shows part of the inheritance hierarchy of the GIM domain model. The overall superclass of the domain is the *imperative-domains AST.* The *imperative-design* class models collections of imperative subprograms. The abstract class *imperative-statement* is the superclass for all imperative-programming statements modeled in the GIM. The *imperative-data-construct* class is the superclass of imperative expressions, data types, and variables modeled in the GIM.

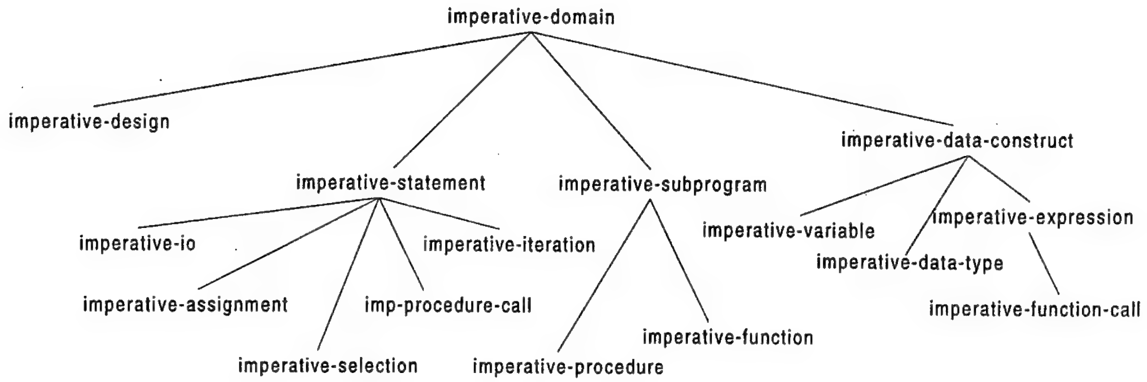


Figure 3: GIM Domain Model Inheritance Hierarchy

Although the Generic Imperative Model (GIM) was developed to model the variables, expressions, assignment statements, and control flow typically built into imperative programming languages, it has some restrictions. These restrictions will be described in Chapter III.

My research develops a transformation system to abstract a COBOL code implemented information up to the design level. At the design level, the extracted information is represented using the Sward's Generic Imperative Model as a generic language.

Transformation Systems

As a possible solution for the maintenance and improvement problem in legacy systems, many efforts based on transformation systems have been proposed to extract the design from the implemented code. This section discusses some of these efforts based on transformation systems that focus on abstracting COBOL code up to the design level.

TAMPR transformation system.

Harmer, McParland and Boyle's research had proposed a transformation system called TAMPR that derives a COBOL program with improved structure from an existing COBOL program [6]. Their work can be seen from the re-engineering context as restructuring at the implementation level. The re-coded program maintains the same behavior as the original program, but with a rationalized structure that facilitates its understanding, thereby reducing the cost of its maintenance.

The TAMPR transformation system was used to build the restructuring transformation system that restructures COBOL programs. This restructuring transformation system receives as input an unstructured COBOL program, and outputs a structured COBOL program. This restructuring transformation system consists of a sequence of transformations built in two phases, each one using specific COBOL language knowledge.

The first phase is based on COBOL Execution Model knowledge. The goal of the Execution Model Stage is to transform a conventional COBOL input program into an output program whose execution behavior is simpler to understand than the model required by COBOL and will facilitate further processing. To achieve this goal, the authors define a simple language form into which all COBOL forms must be converted. This language form requires that:

- procedures be used instead of paragraphs and sections to group sequences of statements;
- procedures have conventional behavior, i.e. they are invoked by stating the procedure name, and when execution terminates they return to the caller;

- goto statements transfer execution control only within the body of a procedure.

The second phase is based on structured programming knowledge. This phase uses the output of COBOL execution model derivation as its input. The structured programming phase transformations are concentrated on improving program structure. The goals of the structuring transformations are:

- to impose a top-down structure on the program;
- to uncover structured constructs expressed by goto statements and to introduce the corresponding structured constructs.

To reduce the effort required in developing restructuring transformations, an intermediate language form is used between the input form, COBOL, and the output form, structured COBOL. Finally, dead code (code that will never be executed) is identified and removed and the standard intermediate language is transformed back in COBOL.

The authors' work has the advantage of being a fully automated program transformation that restructures COBOL programs, while the transformation system developed in my research requires a structured COBOL program as the input program. On the other hand, the authors' restructuring transformation system only has applicability in restructuring COBOL code into COBOL code, while my research uses a generic language to model the COBOL program at the design level.

Inverse Transformation.

Sneed defines inverse transformation as being the process of retranslating software code back into an application specification [11]. The motivation for Sneed's work is the belief that software can be better altered and enhanced at the conceptual level rather than at the physical level where the maintainer's view is often obstructed by implementation details.

In Sneed's work, COBOL programs are retranslated into an intermediate design schema based on a set of normalized relational tables for the modules, data capsules, and interfaces extracted from the source programs. The mapping of COBOL source code back into a design schema is based on a set of transformation rules obtained by inverting the rules used to generate COBOL programs from the design. The design schema used here is structured and modular. If the software has not been designed and implemented in accordance with structured and modular design rules, it may be necessary to restructure and remodularize it. It is in this respect that reverse engineering and restructuring complement one another as shown in Figure 4.

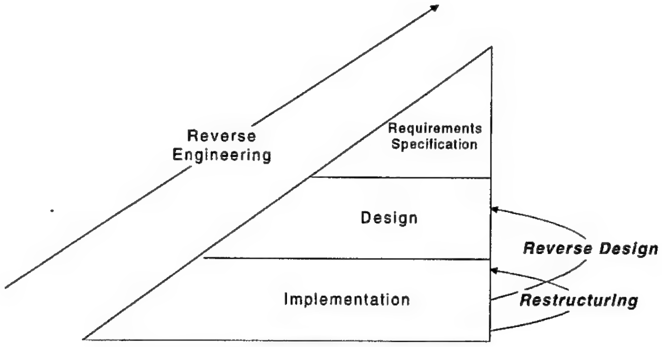


Figure 4. Reverse Engineering and Restructuring

The design schema used to represent the logical level is divided into two major parts: data design, and program design.

The data design part contains five design elements that are:

- database structure design, which provides an overview of the database design;

- file design, which provides information on the characteristics of the files being used;
- data communication design, which provides an implementation-independent view of the system interfaces;
- data capsule design, which provides an implementation-independent view of the data structures;
- data constant design, which identifies all explicitly assigned values in the source programs.

The program design part also contains five design elements. They are:

- process structure design, which depicts the sequential order in which the individual programs are executed;
- component design, which describes characteristics taken from the COBOL Identification Division and provides an overview of the system configuration;
- data flow design, which describes the input/output relationship of the programs to both the data capsules and the data communication;
- module interface design, which captures connections between modules within a program;
- module design, which is a table of pseudo code instructions for each module.

At the end of the retransformation of all programs, the design database contains an aggregated logical description of the source system in the form of normalized relational tables, which can be further processed for documentation, validation, and program description contained in the design of the source language, in this case COBOL. It represents a logical view of the software implementation, equivalent to the ANSI-SPARC logical view of database implementation.

In the author's work, the design extracted from the code can be only used for documentation, validation and program description, different from a design represented in a generic language that can be used to generate code automatically.

COBOL Program Reuse.

Yang, Chu and Sun's research focused on acquiring Entity Relationship models from data intensive source code to obtain reusable COBOL code components and their designs by combining an analysis of data structures and code [7]. The authors' motivation originated from observing the existing related work, in particular Sneed's research reviewed above. Suitable formal transformations for this purpose have been developed, in particular transformations for dealing with COBOL records and files, aliased records, foreign keys, user defined abstract data types, and functional relationships.

Their study on COBOL code reuse focuses on the parts-based approach. In this approach, components are required to be found and understood, and then incorporated into the designed system. Reusable parts are identified through reverse engineering via program transformation. Program understanding is done inside the program transformation process.

The method proposed to reverse engineer COBOL programs into their reusable forms through program transformation is based on a wide spectrum language, RWSL (Reengineering Wide Spectrum Language). This method consists of the following steps:

- translating a COBOL program into RWSL;
- cleaning up the target program in RWSL;
- looking for functionally self-contained modules;
- reverse engineering the modules into their high-level representations in Entity Relationship (ER) diagrams by applying program transformations.

The obtained ER diagrams are viewed as reusable components. The ER diagrams, together with the original code, are used by a Semantic Interface Analysis tool to generate semantic predicates and interface predicates for a reusable module in terms of its pre-conditions, post-conditions and obligations. The abstracted ER diagrams are able to represent the design of the original programs and they make original programs much more understandable, which is a great help to identify reusable COBOL components. The correctness of the obtained ER diagrams is at present checked manually based on human knowledge and expertise. This method only requires source code as its input and it can be applied to heavily modified code typical of systems that have been maintained over many years.

The COBOL code representation in a wide spectrum language obtained after the translations is used by the authors to model the entities and relationships extracted from the COBOL source code. The COBOL constructs irrelevant for acquiring entities and relationships such as *display*, *perform*, *if*, *when*, *until*, and *accept*, are ignored in the authors' work, in contrast to Sward's Generic Imperative Model, which models the fundamental constructs of imperative programming languages.

Draco-PUC Transformational Engine.

Prado Leite, Sant'Anna, and Francisco do Prado have developed a transformational engine Draco-PUC for porting COBOL programs to C++ [8]. Porting occurs in the context of adaptive maintenance, where the functionality stays the same, but there is a need to alter the supporting platform (hardware or software). They argue that if we have the knowledge from the source language and the target language, we are able to specify the mapping between them and to apply this mapping to any program written in the source language.

Draco-PUC is a transformation system. First of all the input program is analyzed to generate a Dast (Draco abstract syntax tree). Then, a set of pre-defined transformation rules are applied by navigating the Dast in a left-to-right, bottom-up fashion. Whenever several options of transformation exist, a rule filter selects among the several options the one to be applied. After the transformation is applied, the new partial Dast is again selected for transformation using the same set of candidate rules, unless there is a change in the control strategy. Once transformations are finished the Dast is pretty-printed. Implementing a porting strategy in Draco requires three basic steps:

- the construction of source and target domain parsers and pretty-printers;
- the construction of auxiliary libraries, which help in bridging the semantic gap between domains;
- the construction of transformation rules that encapsulate the overall strategy.

The authors' work is an extension of the transformational paradigm to use transformations to extract program structure from code, and then to use the produced information to help in the application of porting transformations.

Other Techniques for Extracting Specifications from COBOL Code

This section review some of the papers that focus on using automated tools and intelligent human guidance to reverse COBOL code to its design

Creating Specifications from Code.

Breuer and Lano have developed techniques to reverse COBOL application code back to the design and specification stages to recreate lost information for the application or to extract new information [2]. This method requires a translation of the source language into a structured programming language. The authors' techniques are aimed at representations that have already been restructured to reveal their essential structure. The aim is to structure the code so that each line is a meaningful fragment of a program specification, which explains why gotos must be replaced. These commands contribute only during execution and tend to hide the real function of the program. The output of this reverse engineering process is a formal notation in the Z specification language, where the code becomes supported by a formal description, which may have been previously lacking.

The reverse engineering begins by reorganizing the data and the code of the program to facilitate analysis, aiming to produce well-defined objects and single-function procedures. The final description returned will consist of a set of class definitions, a list of variable declarations using either the classes or the basic types, and a list of descriptions of the functionality of program segments.

Reverse Engineering from COBOL to Narrative Specification.

Yoshino, et al. have described a new reverse engineering technology that generates a narrative specification used by real-world maintainers, which facilitates the understanding of business procedures in existing COBOL programs [14]. It defines business process logic and how to recognize it in a program. It also defines how to generate narrative specifications based on this process logic.

One of the most important results of this research is the determination of what information should be extracted from COBOL programs for software maintenance. To understand programs, software maintainers must understand the program details after restructuring the design logic, which is more abstract than the program code. The process logic recognized from the programs is classified into two types of process: Normal process and Error process, which handles error case procedures. Specifications are constructed in a book paradigm. The book paradigm consists of a preface (or front page), a table of contents, and the body of text, which consists of structured sections and sentences, references, and comments. The preface is the outline of the program, and the body is in the form of narrative sentences with headings. Then, the process logic is extracted from the programs using the following generation techniques:

- error process and normal process - the system recognizes and suggests error and normal process, which helps the maintainers to understand business processes;
- error check process - the system recognizes the branch condition from a normal process to an error process as being an error check process;
- data item editing - the system puts a sequence of assignment statements of the same record together into an assignment table;
- external program invocation - calls to external subprograms provide important logical units to help comprehend the business process.

This system extracts information, which has not been clearly expressed in programs, and creates narrative specification. The system also provides parallel scrolling and proof-correction marking tools based on the information acquired in the specification generation.

Summary.

Chapter II summarized the literature relevant to this research. Chapter III will describe the methodology used to develop the COBOL to the GIM transformation system. III. Methodology.

III. Methodology

Chapter Overview

This chapter describes and explains the methodology used to transform COBOL legacy code into the Generic Imperative Model (GIM) described in Chapter II. First, the methodology is described. Next, the GIM restrictions in modeling some imperative language constructs are discussed. Finally, the COBOL constructs not implement by this work are discussed.

Methodology

The methodology to transform COBOL into the GIM began as a comparative analysis between the structure of a COBOL program and the abstract syntax trees included in the GIM that model an imperative program. The objective of this analysis was to understand what information should be extracted from a COBOL program and what object class in the abstract syntax trees could capture and model extracted COBOL information through the transformation process. To do this, one must start from the components of a COBOL program, - program identification, data, statements, and environment, and then identify equivalent elements and object classes in the GIM. These components are described in more detail as follows.

The Program. A COBOL program consists of four divisions: the Identification Division, Environment Division, Data Division, and Procedure Division. These divisions give information about the program, its connection with the physical environment, its data structures, and its executable statements, respectively. An imperative subprogram in the GIM is an abstraction unit that may be referenced by a name and that groups a sequence of statements with information about the data manipulated by these statements. A GIM subprogram combines information that in COBOL is in four separated divisions. A collection of GIM subprograms constitutes an imperative program.

The Environment. The Environment Division in a COBOL program describes how the program is connected to the physical environment and links internal references to input/output devices to external ones in the particular operating system. The GIM does not model the links between internal and external references, which does not constitute a problem, because the absence of those references does not interfere with the program functionality.

The Data. The Data Division in a COBOL program contains descriptions of data structures both internal and external. External data structures are contained in the File Section or Communication Section, internal data structures are contained in the Working Storage Section, and parameter structures are contained in the Linkage Section. All variables in a COBOL program are declared in the Data Division and they are visible to the entire program. COBOL language does not use actual or formal parameters.

In the GIM, the imperative-data-construct object class is the superclass that models the imperative subprograms' data types and variables, regardless of how they are used, whether they are used as internal data, parameters, or for input or output. GIM subprograms communicate through actual and formal parameters.

The Statements. The Procedure Division in a COBOL program contains the executable statements grouped into *paragraphs*, which in turn may be grouped into units called *sections*. In a typical COBOL program, a single paragraph or a sequence of paragraphs is executed by a perform statement. Such execution transfers control to the first statement of the paragraph and returns to the statement following the perform statement provided the paragraph terminates by completing. Thus, the perform statement plays the role of a call to a subprogram while the statements contained within the paragraphs are the subprogram itself. In the GIM, the imperative-statement is the superclass for all imperative programming statements modeled in the GIM, and includes:

- imperative-assignment, which models an expression evaluation and retention of the result in a variable. It takes the general form:

  ```
  x:= e
  ```

  where x is a variable and e is an expression of the same type;

- imperative-selection, which models selective control flow. It takes the general form

  ```
  If  B  then
          S1
  else
          S2
  ```

  where B is a boolean expression and S1 and S2 are sequences of statements;

- imperative-iteration, which models a control mechanism for repeating a sequence of statements known as iterative control. It takes the general form:

  ```
  While B
          S1
  ```

  where B represents a boolean expression and S1 a sequence of imperative statements;

- imperative-procedure-call, which calls a subprogram by name and passes any actual parameters required by the subprogram;

– imperative-io, which models the imperative languages' input and output statements.

After this brief comparative analysis between the COBOL structure and the GIM, it was possible to conclude that the GIM would be able to represent all COBOL program components, directly or indirectly, except for the information concerning the physical environment. However, while the GIM was developed as a canonical form, where code that performs the same functions has the same format, in the COBOL language several statements with the same meaning can be written in different formats. For example, if one wants to model the sum of three variables, say x, y and z, and store the result of that sum in the variable k, one could use the GIM imperative-assignment object class in the following way:

```
k := x + y + z;
```

However, use of a COBOL add statement would yield at least, three different formats for that operation:

```
add x y z to k          (in this case k has to be initialized with zero)
add x y z giving k
add x y to z giving k
```

This diversity of possible formats allowed by COBOL semantics increases the complexity of the transformation system construction, because that transformation system would have to foresee all possible COBOL variations. One way of reducing such complexity would be to create an intermediate stage by first transforming the source COBOL program in another, equivalent COBOL program. The COBOL code would be rewritten in a homogeneous way that could be called canonical COBOL or restricted COBOL, where statements with the same functionality are codified in the same format.

This transformation of COBOL to canonical COBOL would preserve the original program's meaning according to COBOL semantics. The canonical COBOL code obtained after those transformations would produce the same result of the original code. The canonical COBOL code would be recognized by the same COBOL compiler used for the original COBOL code. In addition, and most important, the canonical COBOL should use constructions similar to those of the GIM, in order to facilitate the transformation of the canonical COBOL into the GIM. For each COBOL construct, the equivalent chosen canonical format is the one closest to the corresponding GIM object class. The closer the GIM object is to the canonical COBOL construct, the less complex it is to directly translate into the GIM.

Once the source code is transformed into canonical COBOL, it would be directly translated into the GIM, if for all canonical COBOL constructs there were a corresponding similar GIM construct. However, some canonical COBOL constructs, such as the perform statement and read statement, do not possess an equivalent or similar GIM construct. Therefore, some COBOL constructs needed more programming effort to be converted into the GIM. This conversion type was called indirect translation. The two stages (transformation and directly/indirectly translation) used in the transformation system are shown in Figure 5.

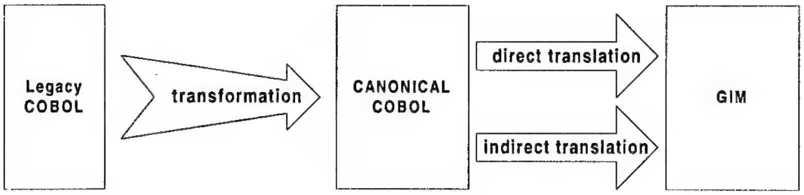


Figure 5. COBOL Code Transformation and its Translation

The next section of this chapter will discuss in more detail some restrictions the GIM presents in modeling an imperative language.

GIM Restrictions

The GIM imposes some restrictions in modeling some imperative language constructs. While some of those restrictions are not applicable to the COBOL language, other must be treated in the construction of COBOL into the GIM transformation system.

Restriction 1:

A formal parameter of a procedure must not be both an input and an output parameter.

This restriction is applicable to COBOL, since the collection of the subprograms/procedures that model the GIM imperative-AST, have their definitions extracted from performs statements and from called subprograms. In both cases, all variables are global. Therefore, once converted in actual parameters, they should be both input and output parameters. In order to avoid violating this restriction, the "process for converting procedures with a parameter that is both input and output parameter into a procedure that has no such parameters" presented in the Appendix D of Sward's dissertation was used.

Restriction 2:

All functions in the GIM return a single value at the end of their execution and have no output parameters.

This restriction is not applicable to COBOL since the COBOL language does not use functions.

Restriction 3:

All actual parameters in subprograms calls must be variables.

This restriction is automatically enforced for COBOL because all parameters in subprogram calls are built during translation to reference only variables in the Data Division.

Restriction 4:

Subprograms to be modeled in the GIM are not allowed to make calls to themselves.

This restriction is not applicable, because in COBOL, a program can be both a called program and a calling program, but can never call itself. Paragraphs are also not allowed to perform themselves.

Restriction 5:

The call tree of a collection of imperative subprograms must be a directed acyclic graph.

This restriction is not applicable, because a COBOL program cannot call any program that directly or indirectly contains it, avoiding the formation of cyclical graphs.

Restriction 6:

All variables in a subprogram are either declared locally or are formal parameters of the subprogram.

This restriction is applicable to COBOL and is enforced by the translation system. The imperative subprogram that corresponds to the body of the main program has only local variables. The imperative subprograms originating from calls and performs have their variables treated as formal parameters.

Restriction 7:

Subprograms cannot be declared inside of another subprogram. They are all declared in the main program's global scope.

This restriction is not applicable to COBOL programs, where all the subprograms are declared out of the main program scope.

Restriction 8:

The GIM does not model heterogeneous data structures.

This was the GIM restriction that had the largest impact on the translation of COBOL into the GIM since most of the data in a COBOL program are structured as records. A specific treatment for this restriction is described in Chapter IV.

Restriction 9:

The GIM does not model pointers.

This restriction is not applicable to COBOL, since COBOL does not use pointer data types.

COBOL Constructs Not Implemented

Some COBOL constructs are not handled by the GIM. Some of them could had been built combining other constructs modeled by the GIM. However that was not done because it would increase the complexity of the system substantially and also because most of them do not interfere in the program's functionality or were not available in the early COBOL language versions. This section presents these constructs in four groups with common characteristics and a brief explanation about them

Group 1.

This group is constituted by statements that had to be manually eliminated from the source code before the transformations.

Go To. The go to statement overrides the normal, sequential execution of statement by transferring control to a designated paragraph or section.

Exit. The exit statement provides a common ending point for a group of procedures usually to leave a procedure before it has finished.

Copy. The copy statement inserts a library text into the source program at compile time.

The go to and exit statements were substituted by perform and if statements while the copy statement was eliminated by inserting the library text in the source program.

Group 2.

This group is constituted by the statements that do not interfere with the program functionality but require additional knowledge from the source code to understand the program at the design level modeled by the GIM.

Merge. The merge statement combines two or more files according to a set of specified keys and makes the records available, in merged order, to an output file or procedure.

Sort. The sort statement creates a sort file making the records available in sorted order.

Group 3.

This group is constituted by the statements that were not implemented but interfere with the program functionality. If one of these statements is in the source code it has to be eliminated before the transformations.

Delete. The delete statement logically removes a record from a mass storage file.

Evaluate. The evaluate statement determines the value of one or more conditions. Subsequent program action depends on the result. (Implemented only in COBOL 85).

Initialize. The initialize statement sets data items to specified values. (Implemented only in COBOL 85).

Inspect. The inspect statement counts and /or replaces the occurrences of a character or group of characters in a data item. (Implemented only in COBOL 85).

Purge. The purge statement eliminates a partial message from a queue. (Implemented only in COBOL 85).

Receive. The receive statement makes a message, a message segment or part of a message segment available to the program.

Rewrite. The rewrite statement logically replaces a existing record in a mass storage file after a read statement without affecting the file position indicator.

Search. The search statement scans a table for an element that satisfies a specified condition.

Send. The send statement causes a message, a message segment, or part of a message to be release to one or more output queues that are maintained by the message control system.

Group 4.

This group is constituted by the statements do not interfere with the program understanding or its functionality.

Cancel. The cancel statement ensures that the next time a program is called it will be in its initial state.

Enter. The enter statement allows a language other than COBOL to be used in a COBOL source program.

Release. The release statement transfers records to the initial phase of a SORT operation.

Replace. The replace statement replaces source program text. (Implemented only in COBOL 85).

Return. The return statement obtains records from the final phase of a sort or merge operation.

String. The string statement concatenates the whole or partial contents of two or more data items into a single data item.

Use For Debugging. The use for debugging statement identifies those user items that are to be monitored by a debugging procedure.

Group 5.

This group is constituted by statements only used to format reports.

Generate. The generate statement causes the report writer control system to produce a report.

Start. The start statement positions the file position indicator prior to sequential retrieval of records.

Suppress. The suppress statement inhibits the presentation of a report group.

Terminate. The terminate statement causes the Report Writer to complete the processing of a report.

Use Before Reporting. The use before reporting statement specifies a procedure that is executed before a report group is presented.

Summary

Chapter III has defined the methodology used to transform COBOL legacy code into the GIM through transformation of COBOL into canonical COBOL and translation of canonical COBOL into the GIM, has presented the GIM restrictions in modeling imperative languages, and also has pointed out the COBOL constructs not implemented by this work. Chapter IV will describe how the COBOL into canonical COBOL transformation was implemented.

IV. Transformation

Chapter Overview

This chapter describes the transformation of the original COBOL abstract syntax tree (AST) into the canonical COBOL. First, transform programs were defined to convert COBOL constructs into equivalent canonical COBOL constructs. Then, the record elimination to satisfy the GIM restriction of not modeling heterogeneous data types is presented. This transformation system was developed in partnership with Capt. Sonia Rodrigues of the Brazilian Air Force. The programs described in this research complement those described in her research.

Introduction

The COBOL-to-GIM transformation system was built using the Software Refinery^{TM} development environment. Refine/COBOL^{TM} , part of Software Refinery^{TM}, is an interactive workbench that supports reengineering of COBOL legacy systems. Refine/COBOL^{TM} encompasses a domain model and grammar for many dialects of COBOL, including COBOL 74, COBOL 85, IBM COBOL II and a generic COBOL format.

To develop the transformation system, Refine/COBOL was configured to work with the generic format. The generic format does not correspond to any specific COBOL dialect. Rather, it is used to configure Refine/COBOL to handle a range of dialects that have common characteristics.

Refine/COBOL parses COBOL source code and builds abstract syntax trees (ASTs) based upon classes and attributes defined for the COBOL language domain model. For instance, parsing a COBOL add statement of the type

add identifier1 identifier2 identifier3 ***to*** variable-1

yields the abstract syntax tree represented in Figure 6.

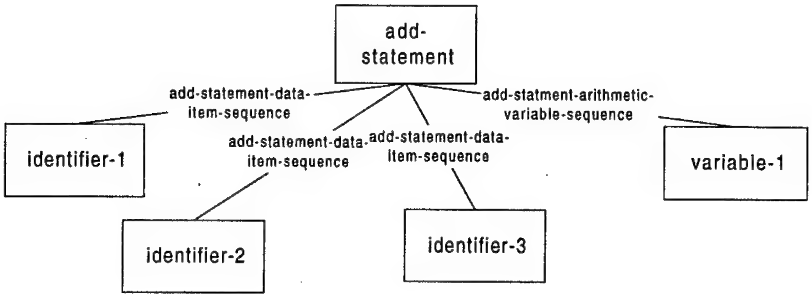

add-statement
add-statement-data-item-sequence
add-statment-arithmetic-variable-sequence
identifier-1
add-statement-data-item-sequence
add-statement-data-item-sequence
variable-1
identifier-2
identifier-3

Figure 6. COBOL Add Statement AST

These ASTs store information about the legacy source code. The transformations modify the original ASTs to conform to canonical COBOL. These transformed COBOL ASTs are the input to the translations that build new GIM ASTs based on those transformed COBOL ASTs.

The objective of transformation is to modify the original COBOL code to conform to canonical COBOL. For each COBOL construct, a program was developed called transform that modifies the original AST to represent canonical COBOL.

Transforms were built using the Refine language transform construct. The Refine transform construct specifies a change to the state of the environment by specifying what conditions the goal state achieves. The transform construct allows specifying side effecting operations by giving the precondition and post condition of the transformation.

To transform COBOL into canonical COBOL, preconditions are defined as patterns that match the original COBOL source code, and post conditions are defined as the canonical COBOL one wants to obtain after the transformations.

Once all the preconditions and their consequent post conditions are defined, the COBOL AST is traversed in a preorder fashion looking for the patterns defined in the preconditions. Every time a precondition is found, the COBOL AST object base representation is modified to satisfy the defined post condition. The following describes the transforms written for this research.

COBOL Add Transform.

The COBOL Add transform, modifies any add statement to its canonical format:

add identifier-1 identifier-2… identifier-n ***giving*** identifier-m.

The COBOL Add transform has two main functions:

(1) Transform-add-to function that transforms add statements of the format:

add identifier-1 identifier-2 … identifier-n ***to*** identifier-1 … identifier-m.

This COBOL add statement is computed by placing the sum of the identifiers before ***to*** in a temporary variable and then adding this temporary variable to the identifier-1, …, identifier-m. The transformation is made by creating one add statement that will store in an auxiliary variable the sum of the identifiers before ***to***, followed by an add statement for each one of the identifiers after ***to***. The creation of an auxiliary variable is necessary to guarantee that the values used in the calculation of the identifiers after ***to***, are the initial values of the identifiers before ***to***, for all new add statements, thereby preserving the program meaning. For instance, let us use the following add statement:

add identiferl identifier-2 identifier-3 ***to*** identifier-1 identifier-4 identifier-5

After the transformation we would obtain:

add identifier-1 identifier-2 identifier-3 ***giving*** variable-auxiliary

add variable-auxiliary identifier-1 ***giving*** identifier-1

add variable-auxiliary identifier-4 ***giving*** identifier-4

add variable-auxiliary identifier-5***giving*** identifier-5

In a COBOL program, all variables must be declared in the Data Division. So, after the transformations, the auxiliary variable that has been created is inserted in the AST that represents the Data Division.

(2) Transform-add-giving function that transforms two formats of add statements:

Format 1: ***add*** identifier-1 identifier-2 ... identifier-n ***giving*** identifier-1 ... identifier-m.

This COBOL add statement is computed by adding together all identifiers before ***giving*** and placing the result into the identifiers after ***giving***. In this case, after the transformation, the first add statement will contain just one identifier after ***giving***. Since all identifiers after ***giving*** will have the same result after the add statement execution, it is possible to use move statements to complete the transformation as seen in the following add statement:

add identiferl identifier-2 identifier-3 ***giving*** identifier-1 identifier-4 identifier-5

In this case, after the transformation we would obtain:

add identifier-1 identifier-2 identifier-3 ***giving*** identifier-1

move identifier-1 ***to*** identifier-4

move identifier-1 ***to*** identifier-5

Format 2: *add* identifier-1 identifier-2 ... identifier-n *to* identifier-n

giving identifier-1 ... identifier-m.

In this case, the identifier after *to* is incorporated into the identifiers before *to,* and then transformed in the same way as the first format.

COBOL Subtract Transform.

The COBOL Subtract transform, modifies any subtract statement to its canonical format:

subtract identifier-1 identifier-2... identifier-n *from* identifier-p *giving* identifier-m.

The COBOL Subtract transform has two main functions:

(1) Transform-sub-from function that transforms subtract statements of the format

subtract identifier-1 identifier-2 ... identifier-n *from* identifier-1 ... identifier-m.

The COBOL subtract statement is computed by placing the sum of the identifiers before *from* in a temporary variable and then subtracting from each one of the identifiers after *from* this temporary variable. The transformation is made by creating an auxiliary variable that will store the sum of the identifiers before *from* and an additional subtract statement for each one of the identifiers after *from.* The creation of an auxiliary variable is necessary to guarantee that the values used in the calculation of the identifiers after *from*, are the initial values of the identifiers before *from*, for all new subtract statements, preserving the program meaning. For instance, let us use the following subtract statement:

subtract identifier-1 identifier-2 identifier-3 ***from*** identifier-1 identifier-4 identifier-5

In this case, after the transformation we will obtain:

add identifier-1 identifier-2 identifier-3 ***giving*** variable-auxiliary

subtract variable-auxiliary ***from*** identifier-1 ***giving*** identifier-1

subtract variable-auxiliary ***from*** identifier-4 ***giving*** identifier-4

subtract variable-auxiliary ***from*** identifier-5 ***giving*** identifier-5

After the transformations the auxiliary variable that has been created is inserted in the AST that represents the Data Division.

(2) Transform-sub-giving function that transforms subtract statements of the format:

subtract identifier-1 identifier-2 … identifier-n ***from*** identifier-p ***giving*** identifier-1 … identifier-m.

In this case, the transformation is made by modifying the original subtract statement to have just one identifier after ***from***, followed by additional move statements for each one of rest of the identifiers after ***from***. For instance, let us use the following subtract statement:

subtract identifier-1 identifier-2 identifier-3 ***from*** identifier-1 ***giving*** identifier-4 identifier-5 identifier-6

In this case, after the transformation we will obtain:

subtract identifier-1 identifier-2 identifier-3 ***from*** identifier-1 ***giving*** identifier-4

move identifier-4 ***to*** identifier-5

move identifier-4 ***to*** identifier-6

COBOL Move Transform.

The COBOL Move transform, modifies any move statement to its canonical format:

move identifier-1 *to* identifier-2.

The COBOL move transform has as its main function, transform-move, that modifies move statements with more than one identifier after *to* transforming it into several move statements with just one identifier after *to*. For instance, let us use the following move statement:

move identifier-1 *to* identifier-2 identifier-3 identifier-4

After the transformation we would obtain:

move identifier-1 *to* auxiliary-variable

move auxiliary-variable *to* identifier-2

move auxiliary-variable *to* identifier-3

move auxiliary-variable *to* identifier-4

The auxiliary variable is necessary to preserve the meaning of move statements of the type:

move identifier-1(index-1) *to* index-2 identifier-2(index-2)

This COBOL move statement is made storing the initial value of identifier-1 (index-1) in a temporary variable and then placing this value into index-2 and identifier-2(index-2).

After the transformations the auxiliary variable that has been created, is inserted in the Data Division Working Storage Section using the same structure of the identifier before *to*.

COBOL Redefines Transform.

In canonical COBOL all data items have just one description entry in the Data Division.

The COBOL Redefines transform has as its main function, transform-redefines, that eliminates all redefined items from the COBOL source code. In a COBOL program, fields contained in a redefining item occupy the same memory positions as the fields being redefined. The redefined items have to be eliminated because the GIM does not handle this construct. To eliminate the redefined items, all operations that refer to these items are modified to refer to the item that is being redefined. After that, the redefined items are eliminated.

Record Elimination.

To avoid violating the GIM restriction of not modeling heterogeneous data types, all group items in a COBOL program had to be transformed in elementary items. Most of the data in a COBOL program are structured as records. A record is a group item of related information uniquely identifiable and treated as a unit. This group item is a data item that is composed of elementary items and/or other group items.

Record elimination is the process of group item transformation into elementary items. This elimination was done in the following stages:

(1) Renaming of all the elementary items. The renaming is made to guarantee the uniqueness of each elementary item and avoid ambiguity when an elementary item is referred, because elementary items belonging to different group items can have the same name.

(2) Mapping group items into their elementary items. This mapping is made to aid in the transformation of statements that manipulate group items into equivalent statements manipulating elementary items. This mapping is made by a function called find-family. The function find-family is a recursive function that retrieves all the descendants of a group item. The descendants of that group item that are elementary items are stored in a sequence called group. For instance, let us use the following group item:

```
01 identifier-1.
        03 identifier -11                       picture x(3).
        03 identifier -12.
                05 identifier -121              picture x(3).
                05 identifier -122.
                        07 identifier -1221     picture x(3).
```

The find-family function we would obtain the sequence:

```
group(identifier -1) = [identifier -11, identifier -121, identifier -1221]
```

(3) Transformation of ***move*** statements that manipulate group items into ***move*** statements manipulating the elementary items that compose that group item.

(4) Transformation of all ***input/output*** statements that refer group items into equivalent ***input/output*** statements referring elementary items as described following for write and accept statements.

COBOL Write Transform. The COBOL write transform has as its main function, transform-write, that modifies write statements with group items transforming it into several write statements with elementary items. The transform-write function calls the find-family function described in item (2) of Record Elimination to determine what elementary items correspond to the group item in the write statement being transformed. For instance, let us suppose the group item identifier-1 from the COBOL If transform example and the write statement:

write identifier-1

After the transformation we would obtain:

write identifier-111

write identifier-1121

COBOL Accept Transform. The COBOL accept transform has as its main function, the transform-accept, that modifies accept statements with group items transforming it into several accept statements with elementary items. The transform-accept function calls the find-family function to determine what elementary items corresponds to the group item in the accept statement being transformed.

For instance, let us use identifier-1 group item from the previous COBOL If transform example and the accept statement:

accept identifier-1 ***from console***

After the transformation we would obtain:

accept identifier-111 ***from console***

accept identifier-1121 ***from console***

(5) Transformation of the original COBOL program eliminating from the Data Division all group items.

Summary

Chapter IV has described how the COBOL into canonical COBOL transformation was made. Chapter V will describe how the canonical COBOL is translated into the GIM and will discuss the ability of GIM to handle COBOL language.

V. Translation and Analysis of the GIM

Chapter Overview

This Chapter describes how the new GIM AST is built, and discusses the ability of GIM to handle COBOL language. First, the translation of canonical COBOL constructs into the GIM is explained. Then, the construction of the new GIM AST is described. Finally, the ability of GIM in modeling COBOL language is analyzed.

Translation of COBOL constructs

This section explains how the canonical COBOL constructs are translated into the GIM. Table 1 summarizes the COBOL constructs translated into the GIM objects classes.

Add Statement.

Add statements have the following format in canonical COBOL:

add identifier-1 identifier-2… identifier-n ***giving*** identifier-m

In this COBOL add statement format, all identifiers preceding ***giving*** are added together. This sum is stored as the new value of the identifier following ***giving***. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-addition object. The add statement format chosen is directly translated into the GIM as the assignment statement:

identifier-m := identifier-1 + identifier-2 + … + identifier-n;

Table 1. COBOL Constructs Translated into the GIM

COBOL statements	GIM Imperative-statements
Add Subtract Multiply Divide Compute Move	Imperative-assignment
Call	Imp-subprogram-call
Accept Read	Imperative-input
Display Write	Imperative-output
Open Close	Imperative-file
If	Imperative-selection
Perform paragraph thru end-paragraph	Imp-subprogram-call
Perform paragraph thru end-paragraph until condition	Imperative-iteration (while not condition) Imp-subprogram-call
Perform paragraph thru end-paragraph varying identifier-1 by identifier-2 until condition	Imperative-assignment Imperative-iteration (while not condition) Imperative-addition Imp-subprogram-call
Perform paragraph thru end-paragraph identifier-1 times	Imperative-assignment Imperative-iteration (while not condition) Imperative-addition Imp-subprogram-call
Main Program (statements delimited by Stop run) Perform (Sequence of statements executed by)	Imperative-subprograms

Subtract Statement.

Subtract statements have the following format in canonical COBOL:

subtract identifier -1 identifier -2... identifier -n ***from*** identifier-m ***giving***
identifier-p

In this COBOL subtract statement format, all identifiers preceding ***from*** are added together. This sum is then subtracted from identifier-n. The result is stored as the new value of identifier-p. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-subtraction object. The subtract statement format chosen is directly translated into the GIM as the assignment statement:

identifier-p := identifier-m -identifier-1 - identifier-2 - ... - identifier-n;

Multiply Statement.

Multiply statements have the following format in canonical COBOL:

multiply identifier-m ***by*** identifier-n ***giving*** identifier-p

In this COBOL multiply statement format, the product of the identifier before ***by*** and the identifier after ***by*** is stored in the identifier following ***giving***. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-multiplication object. The multiply statement format chosen is directly translated into the GIM as the assignment statement:

identifier-p := identifier-m * identifier-n;

Divide Statement.

Divide statements have the following format in canonical COBOL:

divide identifier-m ***by*** identifier-n ***giving*** identifier-p

In this COBOL divide statement the identifier-m is divided by identifier-n and the quotient is placed into identifier-p. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-division object. The divide statement format chosen is directly translated into the GIM as the assignment statement: identifier-p := imperative-name-m / imperative-name-n;

Compute Statement.

Compute statements have the following format in canonical COBOL:

compute identifier-m = arithmetic-expression

In this COBOL compute statement the value determined in the arithmetic-expression is placed into identifier-m. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-expression object. The compute statement format chosen is directly translated into the GIM as the assignment statement:

identifier-m := arithmetic-expression;

Move Statement.

Move statements have the following format in canonical COBOL:

move identifier-m ***to*** identifier-n

In this COBOL move statement, the content of the sending item, identifier-m, is transferred to the receiving variable, identifier-n. The corresponding GIM object class is an imperative-assignment, whose imp-assign-rhs attribute is an imperative-name object. The move statement format chosen is directly translated into the GIM as the assignment statement:

imperative-name-n := imperative-name-m;

Accept Statement.

Accept statements have the following format in canonical COBOL:

accept identifier-n

The accept statement inputs non-file data. It is used to transfer data from a hardware device such as terminal into identifier-n. In this accept statement, the data comes from a standard operating system input device. The accept statement is translated into the GIM as an imperative-input, subclass of imperative-io. The input device is translated into the GIM as the imp-in-logical-file. The accept statement format chosen is directly translated into the GIM because it is similar to the syntax generated by the GIM grammar that has the general form:

read imp-in-logical-file, identifier-1 ... identifier-n;

Read Statement.

Read statements have the following format in canonical COBOL:

read file-name-n

The read statement obtains a record from a file and puts it into the file's record area. Read statements are indirectly translated into the GIM as an imperative-input, subclass of imperative-io. The read statement cannot be translated directly into the GIM, because it only refers to an input file, while the GIM imperative-input also refers to a sequence of identifiers to receive the data read.

The imperative-input is the imperative-io subclass that models accept and read statements. Figure 7 shows the imperative-input object class. For accept statements, the imp-in-logical file models the standard input device where the data comes from, and for read statements it models the name of the file being read. The imp-input-list models the data items being read. For accept statements these data items are translated directly from the accept statement. For read statements these data items are translated from the read-fields attribute. COBOL read gives a logical file while the File Section in the Data Division provides the variables that receive the data. GIM must explicitly mention both. The read-fields-attribute maps the logical file to the variables that receive the data.

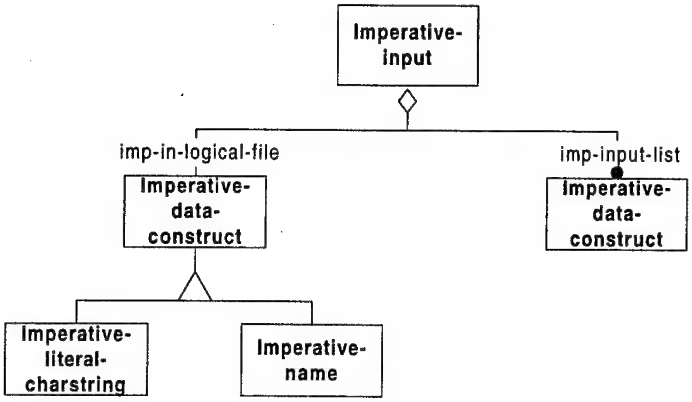

Imperative-input
imp-in-logical-file
imp-input-list
Imperative-data-construct
Imperative-data-construct
Imperative-literal-charstring
Imperative-name

Figure 7. Imperative-Input

Display Statement.

Display statements have the following format in canonical COBOL:

display identifier-1 identifier-2 … identifier-n ***upon*** mnemonic-name

The display statement is used to output non-file data. The display statement is translated into the GIM as an imperative-output, subclass of imperative-io. The display statement format chosen is directly translated into the GIM because it is similar to the syntax generated by the GIM grammar that has the general form:

write imp-out-logical-file, identifier-1 … identifier-n;

Write Statement.

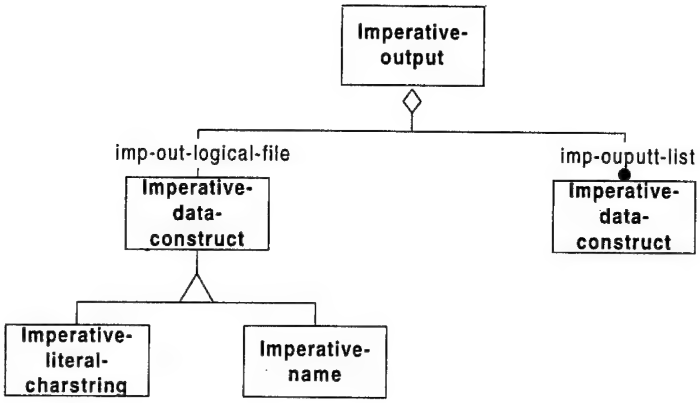
Write statements have the following format in canonical COBOL:

write identifier-n

The write statement writes records to a file or positions lines within a page. The write statement is directly translated into the GIM as an imperative-output, subclass of imperative-io. The write statement format chosen is directly translated into the GIM because it is similar to the syntax generated by the GIM grammar that has the general form:

write imp-out-logical-file, identifier-1 … identifier-n;

The imperative-output is the imperative-io subclass that models display and write statements. Figure 8 shows the imperative-output object class.



Imperative-output
imp-out-logical-file
imp-ouputt-list
Imperative-data-construct
Imperative-data-construct
Imperative-literal-charstring
Imperative-name

Figure 8. Imperative-Output

For display statements, the imp-in-logical file models the mnemonic name defined in the *upon* clause, and for write statements it models the file's name being written. The imp-output-list models the data items being written. These data items are directly translated from the canonical COBOL display and write statements.

Open Statement.

Open statements have the following format in canonical COBOL:

open { ***input/output*** } file-name-n

The open statement makes a file available to the program for processing. The open statement is directly translated into the GIM as an imperative-file. The GIM does not produce grammar to the open statement. The imperative-file is instantiated just to indicate whether the file is available for input or output.

The imperative-file is the imperative-io subclass that models open statements. Figure 9 shows the imperative-file object class. The imp-designator attribute holds name of the logical file as referenced in the open statement. The im-access attribute holds the access type and the imp-status attribute holds the status of the file, opened either for input or output.

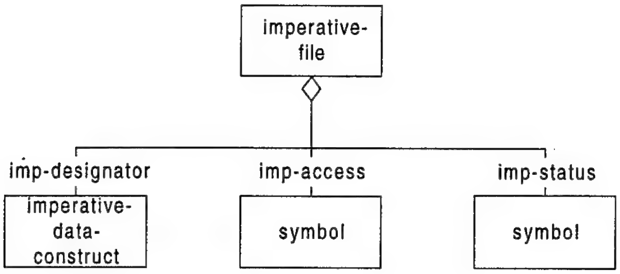


Figure 9. Imperative-File

If Statement.

If statements have the following format in canonical COBOL:

if condition ***then*** statements. { ***else*** statements.}

The if statement evaluates a condition. Subsequent program action depends on whether this condition is true or false. The if statement is directly translated into the GIM as a selection statement:

```
if condition then
        statements;
else
        statements;
endif;
```

The imperative-selection shown in Figure 10 is the GIM object class that models if statements.

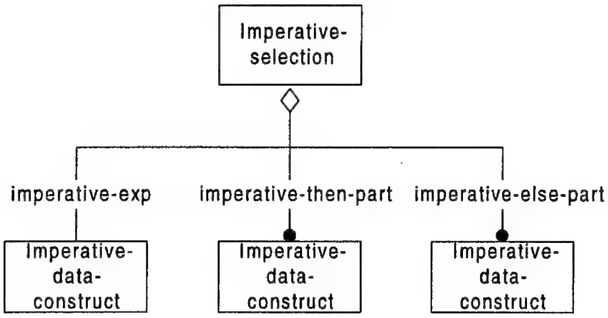

Imperative-selection
imperative-exp
imperative-then-part
imperative-else-part
Imperative-data-construct
Imperative-data-construct
Imperative-data-construct

Figure 10. Imperative-Selection

Call Statement.

Call statements have the following format in canonical COBOL:

call 'subprogram-1' ***using*** identifier-1, identifier-2, ..., identifier-n

In this COBOL call statement, the calling and called programs are linked via shared common data. Both programs' Data Division must contain all data items mentioned in the ***using*** clause. The call statement format chosen is directly translated into the GIM as the procedure call:

subprogram-1 (identifier-1, identifier-2, …, identifier-n);

Perform Statement.

Perform statements have four possible formats in the Canonical COBOL:

Format 1: ***perform*** paragraph-name ***through*** end-paragraph-name

Format 2: ***perform*** paragraph-name ***through*** end-paragraph-name ***until*** condition-1

Format 3: ***perform*** paragraph-name ***through*** end-paragraph-name ***varying*** variable-1 ***from*** identifier-1 ***by*** identifier –2 ***until*** condition-1

Format 4: ***perform*** paragraph-name ***through*** end-paragraph-name identifier-1 ***times***

The GIM does not have a construct to model perform statements, so they are indirectly translated into the GIM as imperative-subprogram-calls. The called imperative-subprogram's name is the paragraph name and the actual parameters are the variables used by the statements executed by the perform statement. These executed statements are translated into imperative-subprograms which formal parameters are equal to the imperative-subprogram-call actual parameters.

For all four formats, the end-paragraph-name is a new paragraph created in the transformation just to delimit the perform scope making it easier to translate. This new paragraph has no statements and its placed just after the last statement executed by the perform statement. For perform statements with a single paragraph, for example, perform paragraph-1, the end-paragraph-name is placed just after the last statement in paragraph-1. For a sequence of paragraphs in the perform statement, for example, perform paragraph-1 through paragraph-2, the end-paragraph-name is placed after the last statement in paragraph-2.

After delimiting the perform scope, formats 2, 3, and 4 have a specific treatment to be indirectly translated into the GIM.

Format 2: ***perform*** paragraph-name ***through*** end-paragraph-name ***until*** condition-1

This perform statement is indirectly translated into a imperative-subprogram-call inside an imperative-iteration that takes the general format:

```
while not condition-1 do
begin
        paragraph-name (parameter-1, ...,parameter-n);
end
```

The condition has to be negated to preserve program's meaning because the statements inside the ***perform until*** are executed while the condition is false, in opposition to the ***while*** statement where the repetition happens while the condition is true.

Format 3: ***perform*** paragraph-name ***through*** end-paragraph-name ***varying*** variable-1 ***from*** identifier-1 ***by*** identifier -2 ***until*** condition-1

This perform statement is indirectly translated into imperative-assignments and into an imperative-subprogram-call inside an imperative-iteration and that takes the general format:

```
variable1:= identifier-1;
while not condition-1 do
begin
        variable-1:= variable-1 + identifier-2 ;
        paragraph-name (parameter-1, ...,parameter-n);
end;
```

Format 4: ***perform*** paragraph-name ***through*** end-paragraph-name

identifier-1 ***times***

This perform statement is indirectly translated into imperative-assignments and into an imperative-subprogram-call inside an imperative-iteration and that takes the general format:

```
variable-1:= 0;
while variable-1 < identifier-1 do
begin
        variable-1:= variable-1 + 1 ;
        paragraph-name (parameter-1, ...,parameter-n);
end
```

The imp-subprogram-call models explicit calls to COBOL subprograms and calls to the subprograms defined from the perform statements. Figure 11 shows the imp-subprogram-call object class.

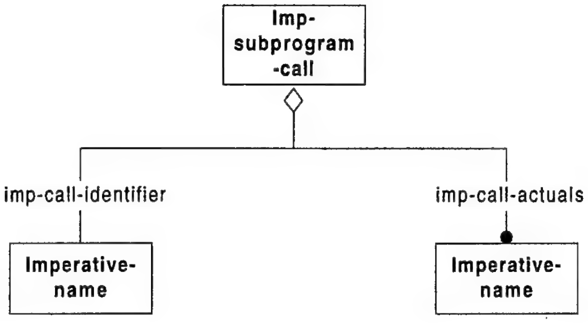


Figure 11. Imp-Subprogram-Call

For explicit calls to a COBOL subprogram the imp-call-actuals are translated from the shared variables referenced in the *using* clause. For calls to subprograms made up from perform statements the imp-call-actuals are translated from the variables referred by the statements executed by these performs.

The Program.

The COBOL main program's body is also translated into the GIM as an imperative-subprogram. The main program consists of all statements before the stop run statement. The COBOL stop run statement closes all files, causes the program to be terminated, and transfers control to the operating system.

An imperative subprogram is composed of an identifier, formal parameters and imperative statements. The first subprogram built in the new GIM AST is the one that corresponds to the main program. Its identifier is the COBOL program ID, and its statements are translated from the statements in the Procedure Division before the stop run statement. Then, the subprograms that corresponds to each one of the perform statements are built. The subprograms extracted from perform statements have as identifier the performed paragraph name. The formal parameters are translated from the variables mapped in the COBOL-symbol-table attribute and their statements are translated from the statements mapped in the statements-table attribute. The COBOL-symbol-table and statements-table attributes will be explained in the next section.

This collection of imperative-subprograms builds the GIM imperative abstract syntax tree (imperative-AST), as shown in Figure 12 using Rumbaugh's notation [10].

Imperative statements comprise the body of each imperative subprogram. These imperative statements are translated from COBOL to the corresponding GIM object to build the new GIM AST.

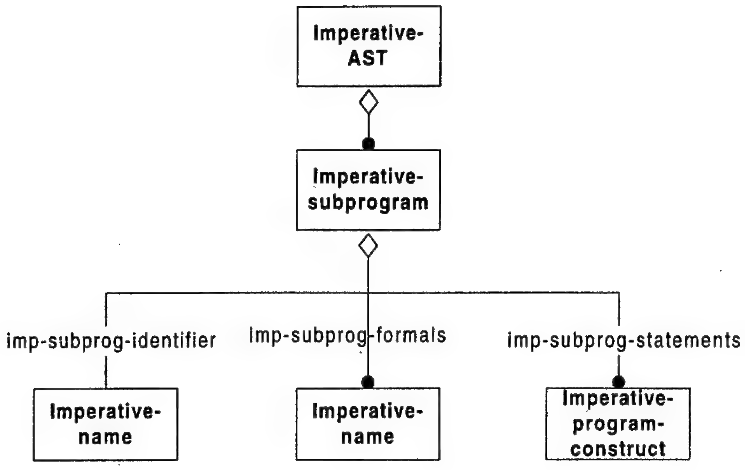


Figure 12. Imperative-AST

Building the new GIM AST

The first approach to translate the COBOL AST into the new GIM AST is to traverse the input COBOL AST and for each visited node to create an object in the new GIM AST whose class models the visited imperative construct. Once all COBOL AST nodes are visited, the new GIM AST can be built.

However, the structures of the COBOL and GIM ASTs are very different, and subsequently caused this approach to be discarded. While a COBOL program has all statements in its Procedure Division, an imperative program is modeled in the GIM as a collection of subprograms. For a COBOL imperative program, these subprograms would be extracted from the main program body delimited by the stop run statement and from the sequence of statements called by a perform statement.

When translating an imperative construct into the GIM, it is necessary that the subprogram of which that statement is part has already been created in the new GIM AST. These subprograms will be the roots of each one of the subtrees that model an imperative program. So, for COBOL to GIM translation, the first step is to build the subprogram corresponding to the COBOL main program, and then traverse the COBOL AST to look for perform statements to complete the collection of imperative subprograms in the GIM AST. Finally the imperative statements for each one of the subprograms are built. To make possible the translation, some additional attributes had to be created in the COBOL AST. These attributes are described next.

New COBOL Attributes.

The attributes described as follows were made to group information that is dispersed throughout the COBOL AST to facilitate extracting it to build the new GIM AST.

Statements-Table Attribute. This attribute maps a perform statement into a sequence of statements. It facilitates knowing what statements are part of the perform statement's scope. For each perform statement visited in the COBOL AST, a new imperative-subprogram is created in the GIM AST. The statements-table attribute supplies the COBOL constructs that are part of that imperative-subprogram.

COBOL-Symbol-Table Attribute. This attribute maps a perform statement into a sequence of variables. It facilitates knowing what variables are used inside the perform statement's scope. In a COBOL program all variables are global, therefore visible to the perform statements. A call to an imperative-subprogram must include actual parameters and a declaration of an imperative-subprogram must include the sequence of formal parameters. The COBOL-symbol-table attribute gives the variables that must be declared as actual and formal parameters.

Expression-Table Attribute. This attribute maps a parent arithmetic expression to its children arithmetic expressions. It facilitates translating the compute statements using a recursive function.

Write-Output Attribute. This attribute maps a write statement to its file description. It facilitates knowing the output file that contains the record referenced in the write statement.

Read-Fields Attribute. This attribute maps a read statement to the data items that composes the file referred in the read statement. It facilitates obtaining the information needed to create the imperative-input object.

Analysis of the GIM

The transformation system developed by this research has shown that almost all COBOL constructs can be modeled by the GIM. However, as addressed in Chapter III, the GIM presents some restrictions when modeling imperative languages. One of these restrictions is that the GIM does not model heterogeneous data types. This restriction has impacted the development of the transformation system, and tends to make GIM programs much larger than their COBOL counterparts.

According to Yang, Chu, and Sun, "COBOL programs represent a class of program in which the complexity is focused on the design and implementation of data structures rather than the algorithms which are often simple. The COBOL language provides complex data structuring mechanisms and high level composite operation to manipulate them. Because of this, COBOL programs are often referred to as data-intensive programs"[7]. Thus, it would be very useful if the GIM could represent records.

Records could be modeled in the GIM by adding imperative-record as a new subclass of imperative-data-type, as shown in Figure 13.

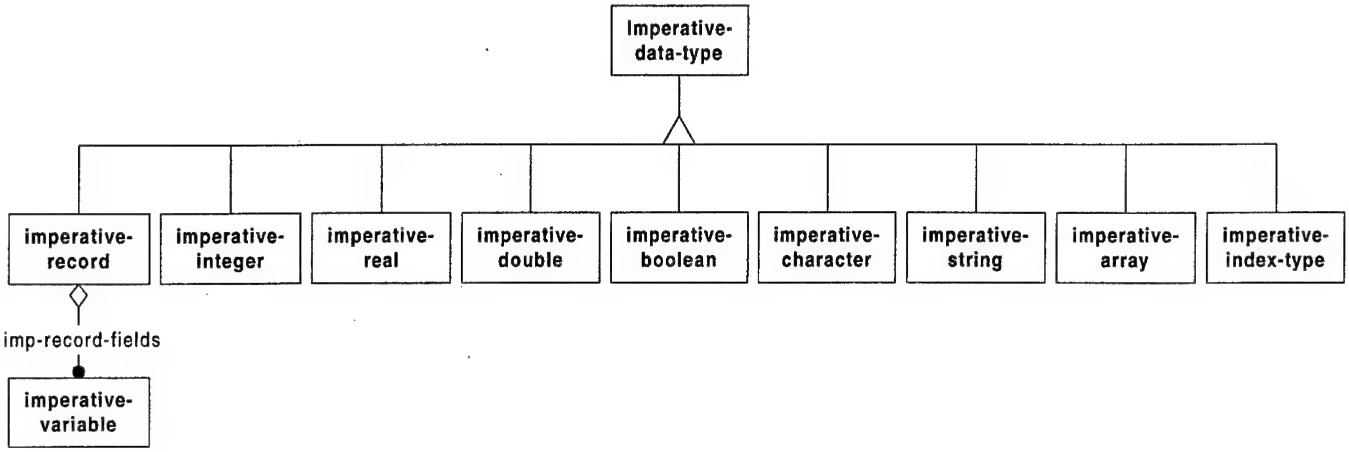


Figure 13. Imperative-Record

Data is modeled in the GIM by two object classes: the imperative-variable, which contains information used to declare the variable, and the imperative-name, which contains information used to refer to the variable. The imperative-variable can support the new data-type imperative-record declaration without changes, but it is not possible to refer to a record field by using the existing imperative-name. Moreover, the current model does not handle array element references correctly, either. The revised imperative-name is detailed in Figure 14 using Rumbaugh's notation. The imp-scope attribute is the imperative-subprogram where the record is referred, the imp-identifier attribute is the record's name or field's name, and the imperative-symbol-table attribute is the reference to the imperative-variable that stores the record declaration; no changes were made to this part. The Imperative-field-reference is the subclass that models a reference to a field of a record type, and the subclass imperative-array-reference models references to array elements.

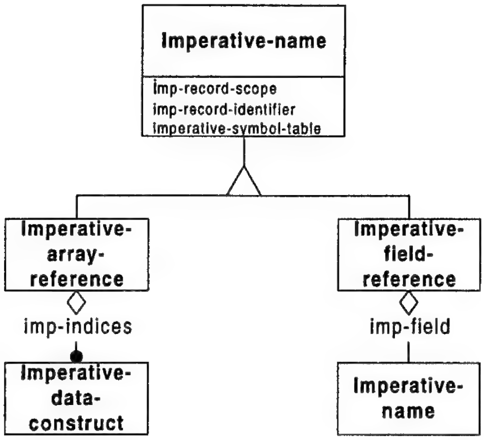


Figure 14. Revised Imperative-Name

Although the GIM is a generic language, it has incorporated some particulars of the FORTRAN language. One of these particularities is the use of implicit data types as declaration of formal parameters. FORTRAN implicitly types variables beginning in the letters A thorough H or P through Z as real and variables beginning with the letters I through N as integers. This implicit declaration is used by the GIM in the imperative-subprograms formal parameters. These parameters are only referenced by their names, without their data-types. The GIM would be more generic if the imperative-subprogram uses imperative-variables to model formal parameters instead of imperative-names. A new attribute, imp-local-vars, could also be created to model local variables within the subprograms.

Most imperative languages have literal constants with a name, which represent its value. The GIM does model some of these constructs, but COBOL has some predefined constants that are not modeled by the GIM. These constants are: figurative zero, which represents one or more instances of the character zero; figurative space, which represents one or more instances of the space character; figurative high-value, which represents one or more instances of the character that has the highest ordinal position in the collating sequence; figurative low-value, which represents one or more instances of the character hat has the lowest ordinal position in the collating sequence; and figurative quotes, which represents one or more instances of the quotation character [12].

The figurative constants zero, space, and quotes could be translated into the GIM as instances of imperative-literal-constant with the values zero, space and quotes respectively. To translate the figurative low-value and high value, however, it is necessary to know the minimum and maximum value that each variable can store in the assignments of low-values or high-values to them. This could be done by adding new attributes imp-value-lower-bound and imp-value-upper-bound to the numeric imperative-data-types and instantiating them in the variables declaration. The new attributes to model minimum and maximum value that a numeric type can store are shown in Figure 15.

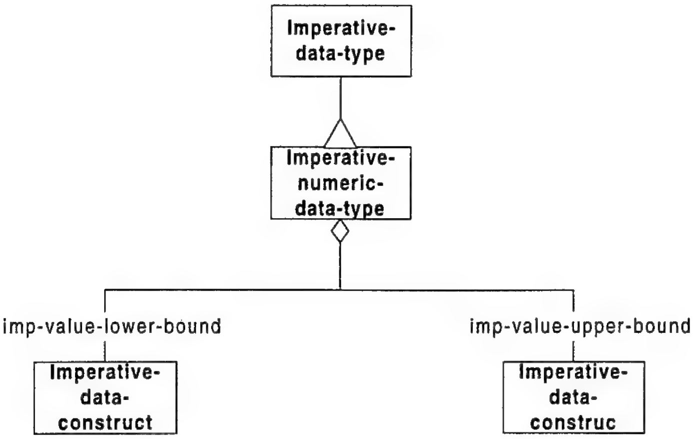


Figure 15. Imperative-Data-Type

With these new attributes, it is not necessary to have the imp-type-size attribute.

Other COBOL constructs not modeled by the GIM are the *sort* statement and the *merge* statement. A *sort* statement causes a COBOL program to process one or more existing unordered files and to produce an ordered file. A *merge* statement causes a COBOL program to process two or more existing ordered files, and to produce from them a single file ordered in the same sequence as the input files.

Applications developed in COBOL often require the sorting of records into a specific order. As the algorithm to process the input file is written according to this order, the knowledge about it is fundamental in the program understanding. Both sort and merge statements process input files and produce the same result: an ascending or descending sorted file that matches a logical order. The sequence in which the produced file is sorted is determined by one or more data items specified in the sort statement *key* clause. These data items are interpreted from left to right as being of decreasing significance.

The GIM imperative-file object class, subclass of imperative-io, could be modified to accommodate this information. The imp-file-name attribute represents the sequence of imperative files being sorted. The imp-order-mode attribute says if the file is sorted in ascending or descending order, and the imp-order-key attribute models the data items which are the keys used in the sort. The *sort* and *merge* statements could be translated into the GIM by instantiation of these new attributes. The *release* and *return* statements that manipulate the sorted files could be translated as imperative-input and imperative-output, sub classes of imperative-io. These statements have the logical functions of read and write file records. Figure 16 shows the imperative-file with these modifications.

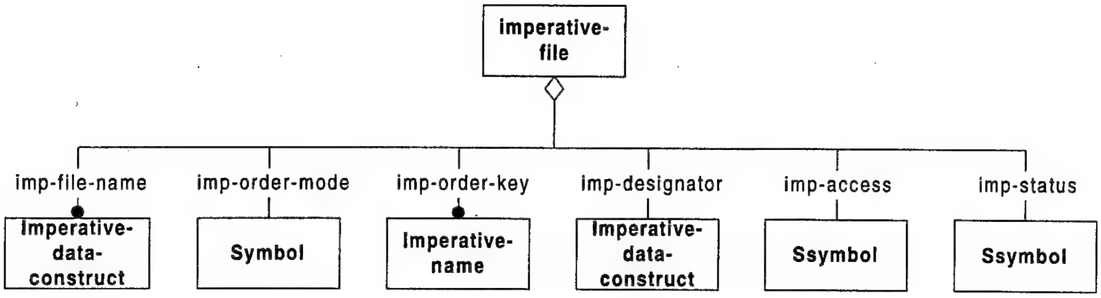


Figure 16. Imperative-File Representing Ordered Files

Summary

Chapter IV has described how the canonical COBOL constructs are translated into the GIM and has discussed the ability of GIM to handle COBOL language. Chapter VI will present the contributions of this study along with suggestions for future research.

VI. Conclusion

Analysis of the Results

This research has developed a COBOL-to-GIM transformation system using Software Refinery^{TM} software that demonstrates that the GIM domain model provides sufficient canonical constructs to model legacy COBOL programs effectively. The diversity of COBOL constructs is modeled by the GIM with generic constructs that can be easily translated to any imperative language. Tables 2 shows the COBOL constructs translated and Table 3 shows those not implemented by this work.

The COBOL-to-GIM transformation system models a COBOL program as a collection of imperative-subprograms. These imperative-subprograms correspond to the body of the main COBOL program (the statements executed before the *stop run* statement,) and to each one of the sequences of statements executed by a *perform*. The imperative-subprograms are organized in a hierarchical structure, where a main control subprogram directs the execution of the others subprograms, which makes the control flow structure clearer and consequently more easily understood.

The global variables manipulated by a sequence of paragraphs are translated into formal parameters of the imperative-subprograms which allows each subprogram to be analyzed separately from the main program. Additionally, paragraphs found after the *stop run* statement that are never executed by a *perform*, are discarded in the transformation, avoiding the creation of dead code within the imperative-subprograms.

Table 2. COBOL Constructs Translated

COBOL statements	GIM Imperative-statements
Add Subtract Multiply Divide Compute Move	Imperative-assignment
Call	Imp-subprogram-call
Accept Read	Imperative-input
Display Write	Imperative-output
Open Close	Imperative-file
If	Imperative-selection
Perform paragraph thru end-paragraph	Imp-subprogram-call
Perform paragraph thru end-paragraph until condition	Imperative-iteration (while not condition) Imp-subprogram-call
Perform paragraph thru end-paragraph varying identifier-1 by identifier-2 until condition	Imperative-assignment Imperative-iteration (while not condition) Imperative-addition Imp-subprogram-call
Perform paragraph thru end-paragraph identifier-1 times	Imperative-assignment Imperative-iteration (while not condition) Imperative-addition Imp-subprogram-call
Main Program (statements delimited by Stop run) Perform (Sequence of statements executed by)	Imperative-subprograms

Table 3. COBOL Constructs Not Implemented

COBOL statements	Treatment
Go To Exit Copy	Not implemented. Treated manually.
Merge Sort	Not implemented. Do not interfere with program functionality.
Delete Evaluate Initialize Inspect Purge Receive Rewrite Search Send Cancel Enter Release Replace Return String Use For Debugging Generate Start Suppress Terminate Use Before Reporting	Not implemented. Not found in the sample legacy source code.

The COBOL-to-GIM transformation system reverse designs a COBOL program, obtaining a modularized design with canonical constructs that captures all the functionality of the original source code. This design can be easily understood, making it possible to restructure it to improve the existing design or to re-design it to extend the functionality and incorporate new enhanced functions.

An example of COBOL code transformation is shown in Appendix A. The example shows the outputs for the two-phased approach. Verification of transformation correctness was done by analyzing and comparing the input COBOL program with the obtained design.

Recommendations

Despite limitations of this study, some recommendations can be offered with regard to the transformation system development approach.

The COBOL-to-GIM transformation system was developed with a two-phased approach. First, the diversity of the COBOL imperative programming language constructs is reduced into a subset of this language by applying transformations, obtaining an equivalent source code with the same meaning but re-coded using homogeneous constructs close to those constructs used by the GIM. Then, this reduced subset is translated into the GIM.

The objective of this approach is to reduce the complexity and programming effort needed in manipulating at the same time two abstract syntax trees (COBOL and GIM) that are completely different. By using this approach, each node of the imperative programming language reduced subset AST is translated into one node of the GIM AST that represents an object class with the same structure of the object class being translated.

The adopted methodology showed its efficacy in reducing the programming effort by working in two steps that manipulate similar structures, which allowed code reuse or minor alterations to implement the translations.

The transformation system developed by this research has shown that almost all COBOL constructs can be modeled by the GIM. However, as addressed in Chapter III, the GIM presents some restrictions when modeling imperative languages.

Future Research

To be modeled by the GIM, a program has to be structured. As the use of *go to* statements in COBOL programs is quite common, future research could address the development of a tool to structure imperative programs as well as to impose on them a top-down, hierarchical structure. A top-down, hierarchical structure can make the execution of the statements inside a paragraph be executed only by a *perform* that refers to this paragraph. By imposing this structure, code duplication generated by the transformation system is avoided. The modifications and the creation of object classes presented in the previous section can also be explored in future research.

Summary

This study explored the GIM as a generic language to model COBOL programs. Representing a COBOL program at a higher abstraction level using a generic language is a fundamental step in the reengineering process. This representation recovers the initial program design, facilitating its documentation, maintenance, making possible the redesign of the program by preserving its original semantic behavior, and adding new functions or extending the existing ones. Once redesigned in the GIM, the canonical constructs can be parameterized with a desired target language and a new, enhanced system can be automatically generated.

Bibliography

1. Bennett, Keith. “Legacy Systems: Coping with Success,” IEEE Software (January 1995).

2. Breuer, P.T. and Lano K. “Creating Specifications from Code: Reverse-engineering Techniques,” Journal of Software Maintenance: Research and Practice , (1991).

3. Byrne, Eric J. “A conceptual foundation for software reengineering,” Proceedings of The International Conference on Software Maintenance. 216-235. IEEE Computer Society Press, Nov 1992.

4. Chikkofsky, Elliott and James H. Cross. “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Software, (January 1990).

5. Corbi, T.A. “Program understanding: Challenge for the 1990s,” IBM Systems Journal (1989).

6. Harmer, Terence J., McParland, Patrick J. and James M. Boyle. “Transformations to Restructure and Re-engineer COBOL Programs,” Automated Software Engineering, (1998).

7. Hongji Yang, William C. Chu, and Young Sun. “A Practical System of COBOL Program Reuse for Reengineering,” IEEE Eighth International Workshop on Technology and Engineering Practice incorporating Computer Aided Software Engineering, (1997).

8. Julio Cesar Sampaio do Prado Leite, Marcelo Sant’Anna, and Antonio Francisco do Prado. “Porting COBOL programs using a transformational approach,” Journal of Software Maintenance: Research and Practice, (1997).

9. Mark van den Brand, Paul Klint, and Chris Verhoef. “Reengineering needs generic programming language technology,” Technical Report P9618, University of Amsterdam, Programming Research Group, (1996).

10. Rumbaugh, James and Michael Blaha. Object-Oriented Modeling and Design. New Jersey, Prentice Hall, Inc., 1991.

11. Sneed, H.M. and Jandrasics G. “Inverse Transformation of Software from Code to Specification,” IEEE Conference on Software Maintenance, (1988).

12. Sordillo. Donald A. The Programmer’s ANSI COBOL Reference Manual. New Jersey: Prentice Hall, 1989.

13. Sward R. E. Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, Sep 1997.

14. T. Yoshino, S. Uehara, T. Ookubo, S. Suguta, Y. Hotta, and M. Sonobe. "Reverse engineering from COBOL to narrative specification," IEEE Proceeding. Nineteenth Annual International Computer Software and Application Conference" (1995).

15. Waters, Richard C. "Program Translation via Abstraction and Re-implementation," IEEE Transactions on Software Engineering (Aug 1988).

Appendix A: Example of COBOL Code Transformation

```
000000 ID DIVISION.
000100 PROGRAM-ID. COBOL001
000200 AUTHOR. Rodrigues e Moraes.
000300 INSTALLATION.
000400 DATE-WRITTEN. 28 jul 98.
000500 REMARKS.
000700 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000701 SPECIAL-NAMES.
000702         console is console.
000720     SKIP2
000730 INPUT-OUTPUT SECTION.
000740     SKIP1
000750 FILE-CONTROL.
000760     SKIP1
000770     SELECT SYS0  ASSIGN TO SYS006-ARQ01
000780                  ORGANIZATION IS INDEXED
000781                 ACCESS MODE IS SEQUENTIAL.
000910     SELECT SYS5  ASSIGN TO SYS011-UT-3350-AS-UNIF
000920                  ORGANIZATION IS SEQUENTIAL.
000940     EJECT
000950 DATA DIVISION.
000960 FILE SECTION.
000999     SKIP3
001000 FD  SYS0,
001020       RECORD CONTAINS 1 TO 12488 CHARACTERS,
001050       LABEL RECORDS ARE STANDARD.
001099     SKIP1
001100 01  max_number                          PICTURE 9(03).
001999     SKIP3
006000 FD  SYS5,
006001     RECORD CONTAINS 1 TO 12488 CHARACTERS,
006002       LABEL RECORDS ARE STANDARD.
006010 01  total-output-01                        PICTURE 9(09).
006010 01  total-output-02                        PICTURE 9(09).
400000 WORKING-STORAGE SECTION.
400338 01  total                               PICTURE 9(07)
400339         USAGE IS COMPUTATIONAL, VALUE IS ZERO.
400340 01  total_odd                           PICTURE 9(07)
400341         USAGE IS COMPUTATIONAL, VALUE IS ZERO.
400342 01  temp                               PICTURE 9(07).
400343 01  a                                  PICTURE 9(07).
400344 01  b                                  PICTURE 9(07).
400345 01  c                                  PICTURE 9(07).
400346 01  d                                  picture 9(05).
400346 01  e occurs 10 times                  picture 9(05).
400355 01     f.
400357     05         f11                   PICTURE X(03).
400358     05         f12                   PICTURE X(03).
400370 01     g.
```

```
400372     05        g11                   PICTURE X(03).
400373     05        g12                   PICTURE X(03).
500000 PROCEDURE DIVISION.
500005 paragraph-start.
500010         OPEN INPUT SYS0 .
600001         OPEN OUTPUT SYS5  .
600002     READ SYS0.
600010         ACCEPT total-output-01 .
600017     MOVE f to g.
600018     ADD a  to  a b d.
600020        PERFORM paragraph-1 THRU compute-paragraph-1 UNTIL a > b.
600040         PERFORM paragraph-4 VARYING a FROM b by c UNTIL  a > d .
600050         WRITE total-output-01 .
600060         ACCEPT total-output-01 .
600070         DISPLAY 'End of main program.'  temp UPON CONSOLE.
600080         STOP RUN.
600090 paragraph-1.
600100          MULTIPLY a BY c GIVING e(1) e(2) e(3).
600030     PERFORM paragraph-2 thru paragraph-3.
600110          COMPUTE a d = b + c  .
600115 compute-paragraph-1.
600120      DISPLAY 'The maximum number is :' max_number .
600150      IF a < b THEN
600160             ADD a b c GIVING d e(1) e(2) e(3).
600170       IF  a > b THEN
600180             DIVIDE c  BY  b  GIVING a d.
600140 paragraph-2.
600140           SUBTRACT a b FROM  c d.
600200 paragraph-3.
600210     MOVE false to a.
600220         WRITE total-output-02 .
600230         ACCEPT total-output-02 .
600250 paragraph-4.
600260      IF a > b THEN
600270            ADD e(a) b  d  c 3 GIVING e(b) e(1) e(2)
600280      ELSE .
600290            MOVE 2.7 to b.
600320          ADD e(a) b  d  c 3 GIVING e(b)  .
```

Figure 17. COBOL Code Before Transformation

Figure 17 shows a COBOL code before the transformation.

```
IDENTIFICATION DIVISION.PROGRAM-ID.COBOL001.
  AUTHOR.
    Rodrigues e Moraes.
  INSTALLATION.
  DATE-WRITTEN.
    28 jul 98.
  REMARKS.
  ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.SPECIAL-NAMES.CONSOLE IS CONSOLE.
    INPUT-OUTPUT SECTION.
      FILE-CONTROL.
        SELECT SYS0 ASSIGN TO SYS006-ARQ01
          ORGANIZATION IS SEQUENTIAL ACCESS MODE IS SEQUENTIAL.
        SELECT SYS5 ASSIGN TO SYS011-UT-3350-AS-UNIF
          ORGANIZATION IS SEQUENTIAL.
  DATA DIVISION.
    FILE SECTION.
      FD SYS0 RECORD CONTAINS 1 TO 12488 CHARACTERS
        LABEL RECORD IS STANDARD.1 MAX_NUMBER PIC IS 9(03).
      FD SYS5 RECORD CONTAINS 1 TO 12488 CHARACTERS
        LABEL RECORD IS STANDARD.
        1 TOTAL-OUTPUT-01 PIC IS 9(09).
        1 TOTAL-OUTPUT-02 PIC IS 9(09).
    WORKING-STORAGE SECTION.
      1 TOTAL PIC IS 9(07) USAGE IS COMPUTATIONAL VALUE IS ZERO.
      1 TOTAL_ODD
        PIC IS 9(07) USAGE IS COMPUTATIONAL VALUE IS ZERO.
      1 TEMP PIC IS 9(07).
      1 A PIC IS 9(07).
      1 B PIC IS 9(07).
      1 C PIC IS 9(07).
      1 D PIC IS 9(05).
      1 E PIC IS 9(05) OCCURS 10 TIMES.
      1 F11-F PIC IS X(03).
      1 F12-F PIC IS X(03).
      1 G11-G PIC IS X(03).
      1 G12-G PIC IS X(03).
      1 VAR-41 PIC IS 9(07).
      1 VAR-42 PIC IS 9(07).
  PROCEDURE DIVISION.
    PARAGRAPH-START.
      OPEN INPUT SYS0.OPEN OUTPUT SYS5.READ SYS0 END-READ.
        ACCEPT TOTAL-OUTPUT-01.MOVE F11 TO G11 MOVE F12 TO G12.
        MOVE A TO VAR-41
        ADD VAR-41 A GIVING A
        ADD VAR-41 B GIVING B
        ADD VAR-41 D GIVING D.
        PERFORM PARAGRAPH-1 THROUGH end_COMPUTE-PARAGRAPH-1
          UNTIL A >B END-PERFORM.
        PERFORM PARAGRAPH-4 THROUGH end_PARAGRAPH-4
          VARYING A FROM B BY C UNTIL A >D END-PERFORM.
        WRITE TOTAL-OUTPUT-01 END-WRITE.
        ACCEPT TOTAL-OUTPUT-01.
        DISPLAY End of main program. UPON CONSOLE
        DISPLAY TEMP UPON CONSOLE .
        STOP RUN.
    PARAGRAPH-1.
  MULTIPLY A BY C GIVING E ( 1)
  MOVE E ( 1) TO E ( 2)
  MOVE E ( 1) TO E ( 3).
    PERFORM PARAGRAPH-2 THROUGH end_PARAGRAPH-3 END-PERFORM.
    COMPUTE A = B + C MOVE A TO D.
COMPUTE-PARAGRAPH-1.
  DISPLAY The maximum number is : .
    IF A <B THEN
      ADD A B C GIVING D
      MOVE D TO E ( 3)
      MOVE D TO E ( 2)
      MOVE D TO E ( 1)
      END-IF.
    IF A >B THEN DIVIDE C INTO B GIVING A MOVE A TO D END-IF.
end_COMPUTE-PARAGRAPH-1.
PARAGRAPH-2.
  ADD A B GIVING VAR-42
  SUBTRACT VAR-42 FROM C END-SUBTRACT
  SUBTRACT VAR-42 FROM D END-SUBTRACT.
PARAGRAPH-3.
  MOVE FALSE TO A.
  WRITE TOTAL-OUTPUT-02 END-WRITE.
    ACCEPT TOTAL-OUTPUT-02.
end_PARAGRAPH-3.
PARAGRAPH-4.
  IF A >B THEN
    ADD E ( A) B D C 3 GIVING E ( B)
    MOVE E ( B) TO E ( 2)
    MOVE E ( B) TO E ( 1)
    ELSE MOVE 2.7 TO B END-IF.
    ADD E ( A) B D C 3 GIVING E ( B).
end_PARAGRAPH-4.
```

Figure 18. COBOL code After Transformation

Figure 18 shows the canonical COBOL obtained after the transformation

```
procedure COBOL001
  ( TOTAL, TOTAL_ODD, TEMP, A, B, C, D, E, F11-F, F12-F,
    G11-G, G12-G, VAR-41, VAR-42, MAX_NUMBER, TOTAL-OUTPUT-01,
    TOTAL-OUTPUT-02
  ) begin
  read ( SYS0, MAX_NUMBER);
  read ( FROM-CONSOLE, TOTAL-OUTPUT-01);
  G11-G := F11-F;
  G12-G := F12-F;
  VAR-41 := A;
  A := VAR-41 + A;
  B := VAR-41 + B;
  D := VAR-41 + D;
  while not A > B do begin
    PARAGRAPH-1 ( C, A, E, B, D, TOTAL-OUTPUT-02) end;
  A := B;
  while not A > D do begin
    PARAGRAPH-4 ( B, E, A, D, C); A := A + C end;
  write ( SYS5, TOTAL-OUTPUT-01);
  read ( FROM-CONSOLE, TOTAL-OUTPUT-01);
  write ( STD-OUTPUT, "End of main program.");
  write ( STD-OUTPUT, TEMP)
  end
procedure PARAGRAPH-1 ( C, A, E, B, D, TOTAL-OUTPUT-02
  ) begin
  E ( 1) := A * C;
  E ( 2) := E ( 1);
  E ( 3) := E ( 1);
  PARAGRAPH-2 ( C, A, D, TOTAL-OUTPUT-02);
  A := B + C;
  D := A;
  write ( STD-OUTPUT, "The maximum number is :");
  if A < B
 then D := A + B + C; E ( 3) := D; E ( 2) := D; E ( 1) := D
   else endif;
  D := A + B + C;
  E ( 3) := D;
  E ( 2) := D;
  E ( 1) := D;
  if A > B then A := C / B; D := A else endif;
  A := C / B;
  D := A
  end
procedure PARAGRAPH-4 ( B, E, A, D, C ) begin
  if A > B
 then E ( B) := E ( A) + B + D + C + 3;
      E ( 2) := E ( B);
      E ( 1) := E ( B)
   else B := 2.7 endif;
  B := 2.7;
  E ( B) := E ( A) + B + D + C + 3;
  E ( 2) := E ( B);
  E ( 1) := E ( B);
  E ( B) := E ( A) + B + D + C + 3
  end
procedure PARAGRAPH-2 ( C, A, D, TOTAL-OUTPUT-02 ) begin
  VAR-42 := A + B;
```

```
C := C - VAR-42;
D := D - VAR-42;
A := false;
write ( SYS5, TOTAL-OUTPUT-02);
read ( FROM-CONSOLE, TOTAL-OUTPUT-02)
end
```

Figure 19. COBOL Code After Translation

Figure 19 shows COBOL code after translation. The imperative subprograms were translated from the main program and from the statements executed by perform.

Vita

Captain Diná Leite Moraes was born on 19 April 1960 in Rio de Janeiro, Brazil. She graduated from Colégio Marista São José in Rio de Janeiro in 1977. She entered undergraduate studies at Santa Úrsula University in Rio de Janeiro, where she graduated with a Bachelor of Science degree in Civil Engineering in December 1982.

Her first assignment was at Diretoria de Administração do Pessoal in August 1984. In March 1986, she was assigned to Centro de Computação da Aeronáutica do Rio de Janeiro – CCA-RJ. In July 1997, she entered the Graduate Computer Systems program, School of Engineering, Air Force Institute of Technology.

Permanent Address: 4 Raymond Court
New Baden – IL – 62265